
Introduction au
PYTHON
Guide pour Débutants
Les bases en moins d' 1h
Robin PARNET

Introduction au Python : Un Guide pour Débutants

Table des matières

---

1. Introduction

Bienvenue dans ce guide d'introduction à Python. Ce livre a pour but de vous initier aux bases de Python, un langage de programmation puissant et polyvalent.

Que vous soyez débutant ou que vous ayez déjà quelques notions de programmation, ce guide vous aidera à démarrer rapidement. Nous aborderons les concepts fondamentaux et fournirons des exemples pratiques pour vous aider à comprendre et à appliquer ces concepts.

---

2. Installation de Python

Pour commencer à programmer en Python, vous devez d'abord installer le langage sur votre ordinateur.

Installation sur Windows

1. Rendez-vous sur le site officiel de Python : python.org
2. Téléchargez la dernière version stable de Python.
3. Exécutez le fichier d'installation et suivez les instructions. Assurez-vous de cocher la case "Ajouter Python à PATH".

Installation sur macOS

1. Ouvrez le terminal.
2. Utilisez Homebrew pour installer Python : *brew install python3*

Installation sur Linux

1. Ouvrez le terminal.
2. Utilisez votre gestionnaire de paquets pour installer Python : *sudo apt-get install python3*

---

3. Votre Premier Programme Python

Pour écrire et exécuter votre premier programme Python :

1. Ouvrez votre éditeur de texte préféré (par exemple, VS Code, Sublime Text).
2. Créez un nouveau fichier et nommez-le hello.py.

3. Écrivez le code suivant :

```
print("Hello, World!")
```

Enregistrez le fichier et exécutez-le en utilisant la commande : python *hello.py*

Cette simple commande affichera "Hello, World!" sur votre terminal ou console, vous indiquant que votre installation de Python fonctionne correctement.

---

4. Les Variables et Types de Données

Python est un langage dynamique où les variables sont créées lors de l'affectation d'une valeur.

Exemples de variables :

```
x = 10          # Entier
y = 3.14        # Flottant
name = "Alice"  # Chaîne de caractères
is_active = True  # Booléen
```

Chaque type de donnée en Python a des propriétés spécifiques et des méthodes associées.

Les Types de Données :

- Entiers (int) : Représente les nombres entiers. Ex : x = 10
- Flottants (float) : Représente les nombres à virgule flottante. Ex : y = 3.14
- Chaînes de caractères (str) : Représente du texte. Ex : name = "Alice"
- Booléens (bool) : Représente les valeurs True ou False. Ex : *is_active = True*

---

5. Les Opérations de Base

Python supporte les **Opérations arithmétiques** de base :

```
a = 5

b = 2

print(a + b)  # Addition : 7

print(a - b)  # Soustraction : 3

print(a * b)  # Multiplication : 10

print(a / b)  # Division : 2.5

print(a % b)  # Modulo : 1

print(a ** b) # Exponentiation : 25
```

Et les **Opérations Logiques :**

```python
x = True
y = False
print(x and y)  # ET logique : False
print(x or y)   # OU logique : True
print(not x)    # NON logique : False
```

---

6. Les Structures de Contrôle

Les structures de contrôle permettent de gérer le flux de votre programme.

Les structures Conditionnelles :

```python
x = 10
if x > 5:
    print("x est supérieur à 5")
elif x == 5:
    print("x est égal à 5")
else:
    print("x est inférieur à 5")
```

Les Boucles *for* et *while* **:**

La boucle for est utilisée pour itérer sur une séquence (liste, tuple, chaîne de caractères, etc.). La syntaxe de base est :

```
for élément in séquence:
    # bloc de code à exécuter pour chaque élément
```

Par exemple:

```
fruits = ["pomme", "banane", "cerise"]
for fruit in fruits:
    print(fruit)
```

Ici, la boucle affiche chaque fruit de la liste.

Pour itérer un cetain nombre de fois, on utilise souvent la fonction *range()*:

```
for i in range(5):
    print(i)
```

Cela affiche les nombres de 0 à 4.

La boucle while continue de s'exécuter tant qu'une condition est vraie. La syntaxe de base est :

```
while condition:
    # bloc de code à exécuter tant que la condition est vraie
```

Par exemple :

```
i = 0
while i < 5:
    print(i)
    i += 1
```

Ici, la boucle affiche les nombres de 0 à 4.

Différences clés

- Boucle *for* : Utilisée pour itérer sur une séquence prédéfinie.
- Boucle *while* : Utilisée quand la répétition dépend d'une condition qui peut changer à chaque itération.

Ces deux types de boucles permettent d'automatiser des tâches répétitives et de rendre le code plus efficace et lisible.

---

7. Les Fonctions

Les fonctions en Python permettent de regrouper des blocs de code réutilisables et de structurer le programme. Voici une explication concise.

Définir une fonction

Pour définir une fonction, on utilise le mot-clé def suivi du nom de la fonction et des parenthèses. La syntaxe de base est :

```
def nom_de_la_fonction(paramètres):
    # bloc de code
    return valeur_de_retour
```

Par exemple *:*

```
def addition(a, b):
    return a + b


somme = addition(3, 5)
print(somme)  # Affiche 8
```

Ici, saluer est une fonction qui prend un paramètre nom et retourne une salutation.

Appeler une fonction

Pour utiliser une fonction, on l'appelle par son nom suivi de parenthèses, avec les arguments nécessaires à l'intérieur, par exemple :

```
def addition(a, b):
    return a + b

somme = addition(3, 5)
print(somme)  # Affiche 8
```

Avantages des fonctions

1. Réutilisabilité : Le code dans une fonction peut être utilisé plusieurs fois sans duplication.
2. Lisibilité : Les fonctions permettent de diviser un programme en blocs logiques.
3. Maintenance : Les modifications peuvent être faites à un seul endroit.

Les fonctions sont des outils essentiels pour écrire du code Python propre, efficace et modulaire.

---

8. Les Listes et Tuples

Listes

Les listes sont des collections ordonnées et modifiables.

```
fruits = ["apple", "banana", "cherry"]
print(fruits[0])  # Accéder au premier élément : "apple"
fruits.append("orange")  # Ajouter un élément
print(fruits)  # Affiche : ["apple", "banana", "cherry", "orange"]
```

- **Accès aux éléments :**

```
print(ma_liste[0])  # Affiche 1
```

- **Modification :**

```
ma_liste[1] = "nouveau"
```

- **Ajout :**

```
ma_liste.append(4)
```

Tuples
Les tuples sont des collections ordonnées et immuables.

```
dimensions = (800, 600)
print(dimensions[0])  # Accéder au premier élément : 800
```

- **Accès aux éléments :**

```
print(mon_tuple[0])  # Affiche 1
```

- **Immutabilité :** Les éléments ne peuvent pas être modifiés après création.

Les listes sont flexibles et modifiables, tandis que les tuples sont fixes et sécurisés pour les données constantes.

---

9. Les Dictionnaires et Ensembles

Dictionnaires

- Définition : Collections non ordonnées de paires clé-valeur.

- Syntaxe :

```
mon_dict = {"clé1": "valeur1", "clé2": "valeur2"}
```

- Accès aux valeurs :

```
print(mon_dict["clé1"])  # Affiche "valeur1"
```

- Ajout/Modification :

```
mon_dict["clé3"] = "valeur3"
```

Ensembles

- Définition : Collections non ordonnées d'éléments uniques.
- Syntaxe :

```
mon_ensemble = {1, 2, 3, 4}
```

- Ajout :

```
mon_ensemble.add(5)
```

- Suppression des doublons : Automatique lors de l'ajout.

Les dictionnaires sont utilisés pour stocker des associations clé-valeur, tandis que les ensembles sont utiles pour les collections uniques sans ordre particulier.

---

10. La Gestion des Fichiers

En Python, vous pouvez manipuler des fichiers en utilisant les fonctions intégrées *open(), read(), write(), close()* et *with.*

- **Ouverture de fichiers** : Utilisez la fonction *open()* pour ouvrir un fichier. Vous devez spécifier le chemin du fichier et le mode d'ouverture (lecture, écriture, ajout, etc.).
- **Lecture de fichiers** : Utilisez la méthode *read()* pour lire le contenu d'un fichier ouvert. Vous pouvez spécifier le nombre d'octets à lire en argument, sinon il lira tout le contenu.
- **Écriture dans des fichiers** : Utilisez la méthode *write()* pour écrire dans un fichier ouvert. Assurez-vous d'ouvrir le fichier en mode écriture ('w'), sinon vous risquez d'écraser le contenu existant.
- **Fermeture de fichiers** : N'oubliez pas de fermer les fichiers après avoir terminé vos opérations avec eux, en utilisant la méthode *close()*. Cela libère les ressources système associées.
- **Gestion automatique des fichiers** : Utilisez l'instruction *with* avec *open()* pour garantir que les fichiers sont fermés correctement une fois que vous avez terminé de les utiliser. Cela évite les fuites de ressources et rend le code plus lisible.

Voici un exemple simple d'écriture dans un fichier :

```
with open("mon_fichier.txt", "w") as f:
    f.write("Bonjour, monde !")
```

Et un exemple de lecture du même fichier :

```
with open("mon_fichier.txt", "r") as f:
    contenu = f.read()
    print(contenu)
```

11. Introduction à la Programmation Orientée Objet

La programmation orientée objet est un paradigme de programmation qui permet de structurer un programme autour d'objets qui représentent des entités du monde réel. En Python, tout est un objet, ce qui signifie que chaque objet possède des attributs (variables) et des méthodes (fonctions) qui lui sont propres.

Les principaux concepts de la POO en Python sont les suivants :

1. **Classes et objets** : Une classe est un modèle pour créer des objets. Les objets sont des instances de classes. Une classe définit les attributs et les méthodes communs à tous ses objets. Par exemple, une classe Chien pourrait avoir des attributs tels que nom et âge, ainsi que des méthodes telles que *aboie()* et *mange()*.
2. **Attributs et méthodes** : Les attributs sont des variables associées à une classe ou à un objet. Les méthodes sont des fonctions associées à une classe ou à un objet. Les méthodes peuvent être utilisées pour modifier les attributs de l'objet ou effectuer des actions liées à cet objet.
3. **Encapsulation** : L'encapsulation est le fait de regrouper les données (attributs) et les méthodes qui les manipulent au sein d'une même entité (la classe). Cela permet de cacher les détails d'implémentation et de protéger les données contre les accès non autorisés.
4. **Héritage** : L'héritage permet à une classe (appelée classe dérivée ou sous-classe) d'hériter des attributs et des méthodes d'une autre classe (appelée classe de base ou superclasse). Cela favorise la réutilisation du code et la création de hiérarchies de classes.
5. **Polymorphisme** : Le polymorphisme permet à des objets de différentes classes d'être traités de manière uniforme. Cela signifie que les méthodes peuvent agir différemment en fonction du type de l'objet sur lequel elles sont appelées.

Voici un exemple simple de classe en Python :

```
class Chien:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def aboie(self):
        print(f"{self.nom} aboie")

    def mange(self, aliment):
        print(f"{self.nom} mange {aliment}")
```

Et voici comment utiliser cette classe pour créer des objets :

```
chien1 = Chien("Buddy", 3)
chien2 = Chien("Milo", 5)

chien1.aboie()
chien2.mange("croquettes")
```

---

12. Les Modules et Packages

Modules :

Un module est un fichier contenant du code Python. Il peut contenir des définitions de fonctions, de classes et de variables. Un module permet d'organiser le code en le

séparant en différents fichiers pour une meilleure lisibilité et une meilleure réutilisation. Pour utiliser les fonctions et les variables définies dans un module, vous pouvez l'importer dans votre script Python à l'aide de l'instruction import.

Par exemple, si vous avez un fichier contenant le code suivant :

```
# mon_module.py

def dire_bonjour():

    print("Bonjour !")


nom = "Alice"
```

Vous pouvez l'importer dans un autre fichier Python comme ceci :

```
import mon_module


mon_module.dire_bonjour()

print(mon_module.nom)
```

Packages :

Un package est un ensemble de modules organisés dans une structure de répertoires. Il permet de structurer et d'organiser le code en sous-dossiers thématiques. Un package doit contenir un fichier spécial nommé *__init__.py* dans chaque sous-dossier pour être reconnu comme un package Python.

Par exemple, si vous avez une structure de répertoire suivante :

```
mon_package/
    __init__.py
    module1.py
    module2.py
```

Vous pouvez importer les modules de votre script python :

```
import mon_package.module1
import mon_package.module2

mon_package.module1.fonction1()
mon_package.module2.fonction2()
```

ou en utilisant l'importation relative :

```
from mon_package import module1, module2

module1.fonction1()
module2.fonction2()
```

13. Les Exceptions et la Gestion des Erreurs

Les exceptions en Python sont des événements qui se produisent lors de l'exécution du programme et qui interrompent le flux normal des instructions. Elles surviennent généralement lorsque le programme rencontre une erreur. La gestion des exceptions permet de gérer ces erreurs de manière contrôlée et d'éviter que le programme ne se termine de manière inattendue.

Utilisation de *try* et *except* :

La structure *try* et *except* permet de capturer et de gérer les exceptions. Le code susceptible de provoquer une exception est placé dans le bloc *try*, et le code de gestion de l'exception est placé dans le bloc *except*.

```
try:
    # Code susceptible de provoquer une exception
    x = 10 / 0
except ZeroDivisionError:
    # Code à exécuter en cas d'exception
    print("Erreur : division par zéro")
```

Bloc *else* :

Le bloc *else* peut être utilisé pour exécuter du code si aucune exception n'a été levée dans le bloc *try*.

```
try:
    x = 10 / 2
except ZeroDivisionError:
    print("Erreur : division par zéro")
else:
    print("La division a réussi")
```

Bloc finally :

Le bloc *finally* contient du code qui s'exécute toujours, que l'exception ait été levée ou non. Il est souvent utilisé pour libérer des ressources (par exemple, fermer un fichier).

```
try:
    fichier = open("mon_fichier.txt", "r")
    contenu = fichier.read()
except FileNotFoundError:
    print("Erreur : fichier non trouvé")
finally:
    fichier.close()
    print("Fichier fermé")
```

Levée d'exceptions :

Vous pouvez lever (ou déclencher) une exception en utilisant *raise*.

```
def verifier_age(age):
    if age < 0:
        raise ValueError("L'âge ne peut pas être négatif")
    return age


try:
    age = verifier_age(-5)
except ValueError as e:
    print(f"Erreur : {e}")
```

Création de vos propres exceptions :

Vous pouvez définir vos propres exceptions en créant une classe qui hérite de *Exception*.

```python
class MonErreur(Exception):
    pass

try:
    raise MonErreur("Ceci est une erreur personnalisée")
except MonErreur as e:
    print(f"Erreur : {e}")
```

Ces concepts vous permettent de gérer les erreurs de manière robuste et de rendre votre code plus résilient.

---

14. Les Bibliothèques Standard Utiles

Python est livré avec une bibliothèque standard très riche qui fournit des modules et des fonctions pour diverses tâches courantes. Voici quelques-unes des bibliothèques standard les plus utiles :

1. os et sys

Ces modules permettent d'interagir avec le système d'exploitation.

- *os* : Fournit des fonctions pour interagir avec le sytème de fichiers et les variables d'environnement.

```python
import os


# Obtenir le répertoire de travail actuel

print(os.getcwd())


# Lister les fichiers dans un répertoire

print(os.listdir('.'))


# Créer un nouveau répertoire

os.mkdir('nouveau_dossier')
```

- *sys* : Fournit des fonctions et des variables pour manipuler différentes parties de l'environnement d'exécution Python.

```python
import sys

# Obtenir la version de Python
print(sys.version)

# Ajouter un chemin à la liste des chemins de recherche des modules
sys.path.append('/chemin/vers/mon_module')
```

2. math

Ce module fournit des fonctions mathématiques de base.

```python
import math

# Calculer la racine carrée
print(math.sqrt(16))

# Calculer le sinus d'un angle (en radians)
print(math.sin(math.pi / 2))
```

3. datetime

Ce module permet de manipuler les dates et les heures.

```python
import datetime

# Obtenir la date et l'heure actuelles
maintenant = datetime.datetime.now()
print(maintenant)

# Créer une date spécifique
date = datetime.date(2023, 5, 17)
print(date)

# Ajouter un délai de temps
delai = datetime.timedelta(days=10)
nouvelle_date = maintenant + delai
print(nouvelle_date)
```

4. random

Ce module permet de générer des nombres aléatoires et de faire des sélections.

```python
import random


# Générer un nombre aléatoire entre 1 et 10
print(random.randint(1, 10))


# Sélectionner un élément aléatoire dans une liste
liste = ['a', 'b', 'c', 'd']
print(random.choice(liste))
```

5. json

Ce module permet de travailler avec des données au format JSON (JavaScriptObjectNotation).

```python
import json


# Convertir un dictionnaire en chaîne JSON
data = {'nom': 'Alice', 'âge': 25}
json_str = json.dumps(data)
print(json_str)


# Convertir une chaîne JSON en dictionnaire
json_data = '{"nom": "Bob", "âge": 30}'
data = json.loads(json_data)
print(data)
```

6. re

Ce module permet de travailler avec les expressions régulières pour la recherche et la manipulation de chaînes de caractères.

```python
import re


# Rechercher toutes les occurrences d'un motif dans une chaîne
texte = "Bonjour, monde! Bonjour, tout le monde!"
motif = r'Bonjour'
resultats = re.findall(motif, texte)
print(resultats)


# Remplacer toutes les occurrences d'un motif dans une chaîne
nouveau_texte = re.sub(motif, 'Salut', texte)
print(nouveau_texte)
```

7. collections

Ce module fournit des types de données spécialisés comme deque, *Counter*, et *defaultdict*.

```python
from collections import deque, Counter, defaultdict

# Utilisation de deque
d = deque([1, 2, 3])
d.append(4)
d.appendleft(0)
print(d)

# Utilisation de Counter
compte = Counter('abracadabra')
print(compte)

# Utilisation de defaultdict
d = defaultdict(int)
d['a'] += 1
print(d)
```

Ces bibliothèques standard sont très puissantes et peuvent simplifier de nombreuses tâches courantes en programmation Python.

---

15. Projets Pratiques

Quelques projets pour pratiquer vos compétences Python :

1. Calculatrice simple

Créez une calculatrice pour des opérations de base.

```python
def calculatrice():
    operation = input("Choisissez une opération (+, -, *, /) : ")
    num1 = float(input("Entrez le premier nombre : "))
    num2 = float(input("Entrez le second nombre : "))

    if operation == '+':
        print(f"Résultat : {num1 + num2}")
    elif operation == '-':
        print(f"Résultat : {num1 - num2}")
    elif operation == '*':
        print(f"Résultat : {num1 * num2}")
    elif operation == '/':
        print(f"Résultat : {num1 / num2}")
    else:
        print("Opération invalide")
```

2. Jeu de devinette de nombre

Un jeu où l'utilisateur doit deviner un nombre aléatoire.

```
import random


def deviner_nombre():
    nombre_secret = random.randint(1, 100)
    tentative = None


    while tentative != nombre_secret:
        tentative = int(input("Devinez le nombre (entre 1 et 100) : "))
        if tentative < nombre_secret:
            print("Trop bas !")
        elif tentative > nombre_secret:
            print("Trop haut !")
        else:
            print("Bravo, vous avez deviné !")
```

3. Gestionnaire de contacts

Un gestionnaire simple pour ajouter et afficher des contacts.

```
contacts = {}


def ajouter_contact(nom, numero):
    contacts[nom] = numero


def afficher_contacts():
    for nom, numero in contacts.items():
        print(f"{nom}: {numero}")


ajouter_contact("Alice", "123456789")
ajouter_contact("Bob", "987654321")
afficher_contacts()
```

4. Générateur de mot de passe

Génère un mot de passe aléatoire.

```
import random

import string


def generer_mot_de_passe(longueur):

    caracteres = string.ascii_letters + string.digits + string.punctuation

    mot_de_passe = ''.join(random.choice(caracteres) for i in range(longueur))

    return mot_de_passe


print(generer_mot_de_passe(12))
```

5. Analyseur de texte

Écrivez un programme qui analyse un texte fourni par l'utilisateur et affiche le nombre de mots, de phrases, et la fréquence des mots.

6. Jeu de pierre-papier-ciseaux

Écrivez un jeu où l'utilisateur peut jouer contre l'ordinateur à pierre-papier-ciseaux. Affichez le score après chaque partie et donnez la possibilité de rejouer.

7. Convertisseur d'unités

Créez un programme qui convertit des unités de mesure et permettez à l'utilisateur de choisir les unités à convertir.

8. To-do list

Créez une application simple de gestion de tâches qui permet d'ajouter, de marquer comme terminées et de supprimer des tâches. Stockez les tâches dans un fichier pour persister les données.

9. Calculateur d'IMC (Indice de Masse Corporelle)

Écrivez un programme qui demande à l'utilisateur son poids et sa taille, puis calcule et affiche son IMC.

10. Scraper de site web

Utilisez une bibliothèque comme requests et BeautifulSoup pour extraire des données d'un site web. Par exemple, récupérez les titres des articles récents d'un blog.

11. Mini-quiz

Créez un quiz simple qui pose des questions à l'utilisateur, vérifie les réponses et affiche le score à la fin. Les questions et réponses peuvent être stockées dans un fichier JSON.

12. Tic-tac-toe

Écrivez un jeu de tic-tac-toe où deux joueurs peuvent jouer l'un contre l'autre sur le même ordinateur. Affichez le plateau de jeu après chaque mouvement et vérifiez les conditions de victoire.

13. Simulateur de lancer de dés

Créez un programme qui simule le lancer de dés. Permettez à l'utilisateur de choisir le nombre de dés et le type de dés (par exemple, 6 faces, 20 faces).

14. Horoscope quotidien

Écrivez un programme qui affiche un horoscope quotidien basé sur le signe astrologique de l'utilisateur. Vous pouvez utiliser des données prédéfinies ou scraper un site web d'horoscope.

---

16. Conseils et Meilleures Pratiques

- Utilisez des noms de variables et des fonctions clairs

- Lisez et Comprenez le code des autres : Consultez des projets open-source pour améliorer vos compétences.
- Commentez votre code : Utilisez des commentaires pour rendre votre code plus compréhensible.
- Utilisez un environnement virtuel : Gérez vos dépendances avec des environnements virtuels (venv).
- Suivez les conventions de codage : Adoptez les conventions PEP 8 pour rendre votre code plus lisible.
- Écrivez des tests pour vérifier que votre code fonctionne comme prévu.

---

17. Ressources Supplémentaires

- Documentation Officielle : docs.python.org
- Tutoriels en Ligne : Recherchez des tutoriels sur YouTube ou des sites comme Real Python.
- Livres Recommandés : "Automate the Boring Stuff with Python" par Al Sweigart, "Python Crash Course" par Eric Matthes.

---

18. Conclusion

Nous espérons que ce guide vous a aidé à démarrer avec Python. Continuez à pratiquer et à explorer ce langage puissant et flexible. La programmation est une compétence précieuse et en constante évolution, et Python est un excellent point de départ.

Ce guide en Python a été conçu pour aider les développeurs de tous niveaux à améliorer leurs compétences en programmation. En cours de rédaction, j'ai utilisé divers outils et ressources, notamment ChatGPT, pour assurer la qualité et la précision du contenu.

---

19. Annexes

Annexe 1 : Table des Opérateurs

Opérateur	Description	Exemple	Résultat
`+`	Addition de deux nombres ou concaténation de deux chaînes de caractères	`3 + 2` `'Hello' + ' ' + 'World'`	`5` `'Hello World'`
`-`	Soustraction de deux nombres	`5 - 3`	`2`
`*`	Multiplication de deux nombres ou répétition d'une chaîne de caractères	`4 * 2` `'Hello' * 3`	`8` `'HelloHelloHello'`
`/`	Division de deux nombres (résultat flottant)	`10 / 2`	`5.0`
`//`	Division entière (résultat entier)	`10 // 3`	`3`
`%`	Modulo, reste de la division entière	`10 % 3`	`1`
`**`	Exponentiation, élève un nombre à la puissance d'un autre	`2 ** 3`	`8`
`==`	Égalité, vérifie si deux valeurs sont égales	`3 == 3`	`True`
`!=`	Inégalité, vérifie si deux valeurs sont différentes	`3 != 4`	`True`
`>`	Supérieur, vérifie si la première valeur est supérieure à la deuxième	`5 > 3`	`True`
`<`	Inférieur, vérifie si la première valeur est inférieure à la deuxième	`5 < 3`	`False`
`>=`	Supérieur ou égal, vérifie si la première valeur est supérieure ou égale à la deuxième	`5 >= 3`	`True`
`<=`	Inférieur ou égal, vérifie si la première valeur est inférieure ou égale à la deuxième	`5 <= 3`	`False`

`and`	Opérateur logique ET, vérifie si les deux conditions sont vraies	`True and False`	`False`
`or`	Opérateur logique OU, vérifie si au moins une des conditions est vraie	`True or False`	`True`
`not`	Opérateur logique NON, inverse la valeur de vérité	`not True`	`False`
`in`	Vérifie si un élément est présent dans une séquence	`'a' in 'apple'`	`True`
`not in`	Vérifie si un élément n'est pas présent dans une séquence	`'b' not in 'apple'`	`True`
`is`	Vérifie si deux références pointent vers le même objet	`a is b`	Dépend des valeurs de `a` et `b`
`is not`	Vérifie si deux références ne pointent pas vers le même objet	`a is not b`	Dépend des valeurs de `a` et `b`

Annexe 2 : Fonctions intégrées utiles

Fonction	Description	Exemple
`len()`	Renvoie la longueur (le nombre d'éléments) d'un objet tel qu'une liste, une chaîne de caractères, un tuple, etc.	`ma_liste = [1, 2, 3]` `print(len(ma_liste)) # Sortie : 3`
`range()`	Génère une séquence de nombres, souvent utilisée dans les boucles `for`. Peut prendre un, deux ou trois arguments (start, stop, step).	`for i in range(5):` ` print(i) # Sortie : 0, 1, 2, 3, 4`
`print()`	Affiche des objets à l'écran. Peut prendre plusieurs arguments séparés par des virgules et possède des options de formatage.	`print("Bonjour, monde!") # Sortie : Bonjour, monde!`
`type()`	Renvoie le type d'un objet.	`print(type(123)) # Sortie : <class 'int'>`
`int()`	Convertit une valeur en entier.	`print(int("123")) # Sortie : 123`
`float()`	Convertit une valeur en nombre flottant.	`print(float("123.45")) # Sortie : 123.45`
`str()`	Convertit une valeur en chaîne de caractères.	`print(str(123)) # Sortie : '123'`